Cours Programmation I (chapitres 1&2)

Licence Fondamentale SMI (semestre 3)

Pr. Mouad BEN MAMOUN

ben_mamoun@fsr.ac.ma

Année universitaire 2014/2015

Plan du cours (1)

1. Introduction
2. Types de base, variables, constantes
3. Opérateurs et expressions
4. Les entrées-sorties (printf, scanf, …)
5. Les structures de contrôle

Plan du cours (2)

6. Les tableaux
7. Les pointeurs
8. Les fonctions

Langages informatiques

- Un langage informatique est un outil permettant de donner des ordres (**instructions**) à la machine
  - A chaque instruction correspond une action du processeur
- Intérêt : écrire des **programmes** (suite consécutive d'instructions) déstinés à effectuer une tache donnée
  - Exemple: un programme de gestion de comptes bancaires
- Contrainte: être compréhensible par la machine

Langage machine

- Langage **binaire**: l'information est exprimée et manipulée sous forme d'une suite de bits

- Un **bit** (*binary digit*) = 0 ou 1 (2 états électriques)

- Une combinaison de 8 bits= 1 **Octet** ➔ $2^8=256$ possibilités qui permettent de coder tous les caractères alphabétiques, numériques, et symboles tels que ?,*,&, …

  - Le code **ASCII** (*American Standard Code for Information Interchange*) donne les correspondances entre les caractères alphanumériques et leurs représentation binaire, Ex. A= 01000001, ?=00111111

- Les opérations logiques et arithmétiques de base (addition, multiplication, …) sont effectuées en binaire

L'assembleur

- Problème: le langage machine est difficile à comprendre par l'humain

- Idée: trouver un langage compréhensible par l'homme qui sera ensuite converti en langage machine
  - **Assembleur** : exprimer les instructions élémentaires de façon symbolique

ADD A, 4

LOAD B ——traducteur——→ langage machine

MOV A, OUT

…

  - +: déjà plus accessible que le langage machine
  - -: dépend du type de la machine (n'est pas ***portable***)
  - -: pas assez efficace pour développer des applications complexes

$\Rightarrow$ **Apparition des langages évolués**

Langages haut niveau

- Intérêts multiples pour le haut niveau:
  - proche du langage humain «anglais» (compréhensible)
  - permet une plus grande portabilité (indépendant du matériel)
  - Manipulation de données et d'expressions complexes (réels, objets, a*b/c, …)
- Nécessité d'un traducteur (compilateur/interpréteur), exécution plus ou moins lente selon le traducteur

Code source en langage évolué → **Compilateur ou interpréteur** → **Langage machine**

Compilateur/interpréteur

- Compilateur: traduire le programme entier une fois pour toutes



  - \+ plus rapide à l'exécution
  - \+ sécurité du code source
  - \- il faut recompiler à chaque modification

- Interpréteur: traduire au fur et à mesure les instructions du programme à chaque exécution



  - \+ exécution instantanée appréciable pour les débutants
  - \- exécution lente par rapport à la compilation

Langages de programmation:

- Deux types de langages:
  - Langages procéduraux
  - Langages orientés objets

- Exemples de langages:
  - **Fortran, Cobol, Pascal, C, …**
  - **C++, Java, …**

Historique du C

- Le langage C a été conçu en 1972 dans «Bell Laboratories » par *Dennis Ritchie* avec l'objectif d'écrire un système d'exploitation (UNIX).

- En 1978, une première définition rigoureuse du langage C (*standard K&R-C*) a été réalisée par *Kernighan et Ritchie* en publiant le livre «The C Programming Language ».

- Le succès du C et l'apparition de compilateurs avec des extensions particulières ont conduit à sa normalisation.

- En 1983, l'organisme ANSI (American National Standards Institute) chargeait une commission de mettre au point une définition explicite et portable pour le langage C. Le résultat est le *standard ANSI-C*.

Caractéristiques du C

- Universel : n'est pas orienté vers un domaine d'application particulier (applications scientifiques, de gestion, …)
- Près de la machine : offre des opérateurs qui sont proches de ceux du langage machine (manipulations de bits, d'adresses, …) ➔ efficace
- Modulaire: peut être découpé en modules qui peuvent être compilés séparément
- Portable: en respectant le standard ANSI-C, il est possible d'utiliser le même programme sur plusieurs systèmes (hardware, système d'exploitation)

Remarque : Une programmation efficace et compréhensible en C n'est pas facilement accessible à des débutants

Programme source, objet et exécutable

- Un programme écrit en langage C forme un texte qu'on nomme *programme ou code source*, qui peut être formé de plusieurs fichiers sources

- Chaque fichier source est traduit par le compilateur pour obtenir un *fichier ou module objet* (formé d'instructions machine)

- Ce fichier objet n'est pas exécutable tel quel car il lui manque les instructions exécutables des fonctions standards appelées dans le fichier source (printf, scanf, …) et éventuellement d'autres fichiers objets

- *L'éditeur de liens* réunit les différents modules objets et les fonctions de la bibliothèque standard afin de former *un programme exécutable*

Remarque : la compilation est précédée par une phase de prétraitement (inclusion de fichiers en-tête) réalisé par le *préprocesseur*

Compilateurs C

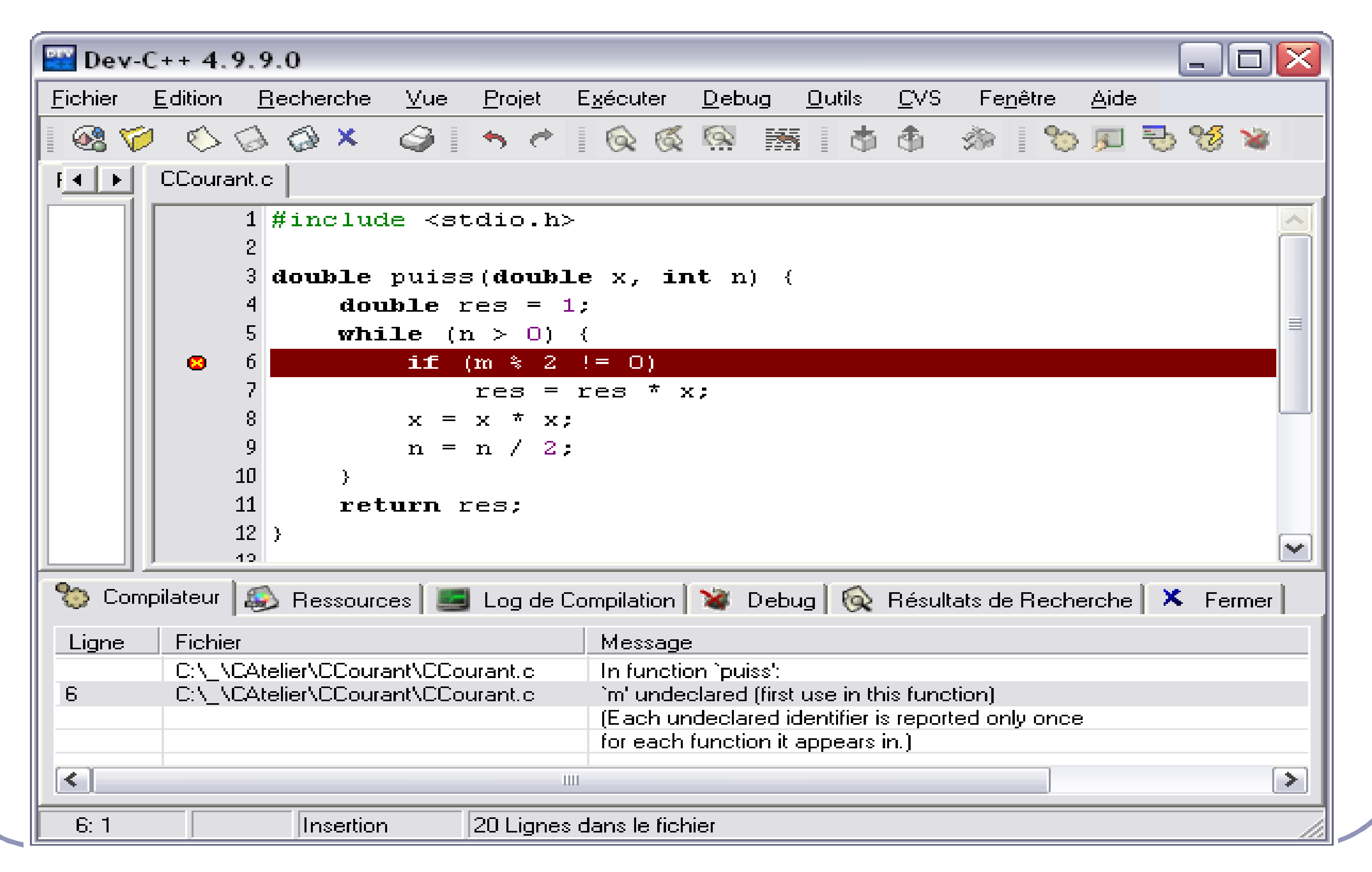
- Pour pouvoir écrire et exécuter des programmes en C, vous avez besoin d’un compilateur C sur votre machine

- Il existe plusieurs compilateurs respectant le standard ANSI-C. Une bonne liste est disponible sur : c.developpez.com/compilateurs/

- En TP, on va utiliser l'environnement de développement Dev-C++ ou Code::Blocks avec le système d’exploitation Windows

- Vous pouvez télécharger Dev-C++ librement, par exemple sur le site www.bloodshed.net et Code::Blocks sur www.codeblocks.org

Exemple d'une fenêtre Dev-C++



Composantes d'un programme C

- Directives du préprocesseur
  - inclusion des fichiers d'en-tête (fichiers avec extension .h)
  - définitions des constantes avec **#define**
- déclaration des variables globales
- définition des fonctions (En C, le programme principal et les sous-programmes sont définis comme fonctions)
- Les commentaires : texte ignoré par le compilateur, destiné à améliorer la compréhension du code

exemple :

```
#include<stdio.h>
main()
{
  printf( "notre premier programme C \n");
 /*ceci est un commentaire*/
}
```

Remarques sur ce premier programme

- *#include<stdio.h>* informe le compilateur d'inclure le fichier stdio.h qui contient les fonctions d'entrées-sorties dont la fonction printf

- La fonction **main** est la fonction principale des programmes en C: Elle se trouve obligatoirement dans tous les programmes. L'exécution d'un programme entraîne automatiquement l'appel de la fonction **main**.

- L'appel de printf avec l'argument "notre premier programme C\n" permet d'afficher : notre premier programme C et \n ordonne le passage à la ligne suivante

- En C, *toute* instruction simple est terminée par un point-virgule **;**

- Un commentaire en C est compris entre // et la fin de la ligne ou bien entre /* et */

Chapitre 2

Variables, types, opérateurs et expressions

Les variables

- Les variables servent à stocker les valeurs des données utilisées pendant l'exécution d'un programme

- Les variables doivent être **déclarées** avant d'être utilisées, elles doivent être caractérisées par :

  - un nom (**Identificateur**)
  - un **type** (entier, réel, …)

(Les types de variables en C seront discutés par la suite)

Les identificateurs

Le choix d'un identificateur (nom d'une variable ou d'une fonction) est soumis à quelques règles :

- doit être constitué uniquement de lettres, de chiffres et du caractère souligné _ (Eviter les caractères de ponctuation et les espaces)
  correct: PRIX_HT, prixHT **incorrect: PRIX-HT, prix HT, prix.HT**
- doit commencer par une lettre (y compris le caractère souligné)
  correct : A1, _A1 **incorrect: 1A**
- doit être différent des mots réservés du langage : **auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while**

Remarque : C distingue les majuscules et les minuscules. NOMBRE et nombre sont des identificateurs différents

Les types de base

- Le type d'une variable détermine l'ensemble des valeurs qu'elle peut prendre et le nombre d'octets à lui réserver en mémoire

- En langage C, il n'y a que deux types de base *les entiers* et *les réels* avec différentes variantes pour chaque type

Remarques:

- Un type de base est un type pour lequel une variable peut prendre une seule valeur à un instant donné contrairement aux types agrégés

- Le type caractère apparaît en C comme cas particulier du type entier (un caractère est un nombre entier, il s'identifie à son code ASCII)

- En C il n'existe pas de type spécial pour chaînes de caractères. Les moyens de traiter les chaînes de caractères seront présentés aux chapitres suivants

- Le type booléen n'existe pas. Un booléen est représenté par un entier (un entier non nul équivaut à vrai et la valeur zero équivaut à faux)

Types Entier

4 variantes d'entiers :

- **char** : caractères (entier sur 1 octet : - 128 **à** 127)
- **short ou short int** : entier court (entier sur 2 octets : - 32768 **à** 32767)
- **int** : entier standard (entier sur 2 ou 4 octets)
- **long ou long int** : entier long (4 octets : - 2147483648 **à** 2147483648)

Si on ajoute le préfixe **unsigned** à la définition d'un type de variables entières, alors la plage des valeurs change:

- **unsigned char** : 0 à 255
- **unsigned short** : 0 à 65535
- **unsigned int** : dépend du codage (sur 2 ou 4 octets)
- **unsigned long** : 0 à 4294967295

Remarque : Une variable du type **char** peut subir les mêmes opérations que les variables du type **short**, **int** ou **long**

Types Réel

3 variantes de réels :

- **float** : réel simple précision codé sur 4 octets de

  $-3.4*10^{38}$ à $3.4*10^{38}$

- **double** : réel double précision codé sur 8 octets de

  $-1.7*10^{308}$ à $1.7*10^{308}$

- **long double** : réel très grande précision codé sur 10 octets de **$-3.4*10^{4932}$ à $3.4*10^{4932}$**

Déclaration des variables

- Les *déclarations* introduisent les variables qui seront utilisées, fixent leur type et parfois aussi leur valeur de départ (initialisation)

- Syntaxe de déclaration en C

  <Type> <NomVar1>,<NomVar2>,...,<NomVarN>;

- Exemple:

```
int   i, j,k;
float  x, y ;
double z=1.5; // déclaration et initialisation
short compteur;
char c=`A`;
```

Déclaration des constantes

- Une constante conserve sa valeur pendant toute l'exécution d'un programme
- En C, on associe une valeur à une constante en utilisant :
  - la directive *#define :*

    #define nom_constante valeur

    Ici la constante ne possède pas de type.

    exemple: *#define Pi 3.141592*

  - le mot clé *const* :

    const *type nom = expression* ;

    Dans cette instruction la constante est typée

    exemple : *const float Pi =3.141592*

(Rq: L'intérêt des constantes est de donner un nom parlant à une valeur, par exemple NB_LIGNES, aussi ça facilite la modification du code)

Constantes entières

On distingue 3 formes de constantes entières :

- forme décimale *:* c'est l'écriture usuelle. Ex : 372, 200
- forme octale (base 8) : on commence par un 0 suivi de chiffres octaux. Ex : 0477
- forme hexadécimale (base 16) : on commence par 0x (ou 0X) suivis de chiffres hexadécimaux (0-9 a-f). Ex : 0x5a2b, 0Xa9f

Remarques sur les constantes entières

- Le compilateur attribue automatiquement un type aux constantes entières. Il attribue en général le type le plus économique parmi (int, unsigned int, long int, unsigned long int)

- On peut forcer la machine à utiliser un type de notre choix en ajoutant les suffixes suivants:
  - u ou U pour unsigned int, Ex : 100U, 0xAu
  - l ou L pour long, Ex : 15l, 0127L
  - ul ou UL pour unsigned long, Ex : 1236UL, 035ul

Constantes réelles

On distingue 2 notations :

- notation décimale Ex : 123.4, .27, 5.
- notation exponentielle Ex : 1234e-1 ou 1234E-1

Remarques :

- Les constantes réelles sont par défaut de type double
- On peut forcer la machine à utiliser un type de notre choix en ajoutant les suffixes suivants:
  - f ou F pour le type float, Ex: 1.25f
  - l ou L pour le type long double, EX: 1.0L

Les constantes caractères

- Se sont des constantes qui désignent un seul caractère, elles sont toujours indiquées entre des apostrophes, Ex : ‘b’, ‘A’, ‘?’
- La valeur d’une constante caractère est le code ASCII du caractère
- Les caractères constants peuvent apparaître dans des opérations arithmétiques ou logiques
- Les constantes caractères sont de type int

Expressions et opérateurs

- Une *expression* peut être une valeur, une variable ou une opération constituée par des valeurs, des constantes et des variables reliées entre eux par des *opérateurs*

  exemples: 1, b, a*2, a+ 3*b-c, …

- Un *opérateur* est un symbole qui permet de manipuler une ou plusieurs variables pour produire un résultat. On distingue :
  - les *opérateurs binaires* qui nécessitent deux opérandes (ex : a + b)
  - les *opérateurs unaires* qui nécessitent un seul opérande *(ex: a++)*
  - *l'opérateur conditionnel ?:* , le seul qui nécessite trois opérandes

- Une expression fournit une seule valeur, elle est évaluée en respectant des règles de priorité et d'associativité

Opérateurs en C

- Le langage C est riche en opérateurs. Outre les opérateurs standards, il comporte des opérateurs originaux d'affectation, d'incrémentation et de manipulation de bits

- On distingue les opérateurs suivants en C :

  - **les opérateurs arithmétiques : +, -, *, /, %**
  - **les opérateurs d'affectation : =, +=, -=,*=,/=,…**
  - **les opérateurs logiques : &&, ||, !**
  - **les opérateurs de comparaison** : **==, !=, <, >, <=, >=**
  - **les opérateurs d'incrémentation et de décrémentation : ++, --**
  - **les opérateurs sur les bits : <<, >>, &, |, ~, ^**
  - **d'autres opérateurs particuliers :** ?:, sizeof, cast

Opérateurs arithmétiques

- binaires : + - * / et % (modulo) et unaire : -
- Les opérandes peuvent être des entiers ou des réels sauf pour % qui agit uniquement sur des entiers
- Lorsque les types des deux opérandes sont différents il y'a conversion implicite dans le type le plus fort
- L'opérateur / retourne un quotient entier si les deux opérandes sont des entiers (5 / 2 ➔ 2). Il retourne un quotient réel si l'un au moins des opérandes est un réel (5.0 / 2 ➔ 2.5)

Conversions implicites

- Les types short et char sont systématiquement convertis en int indépendemment des autres opérandes

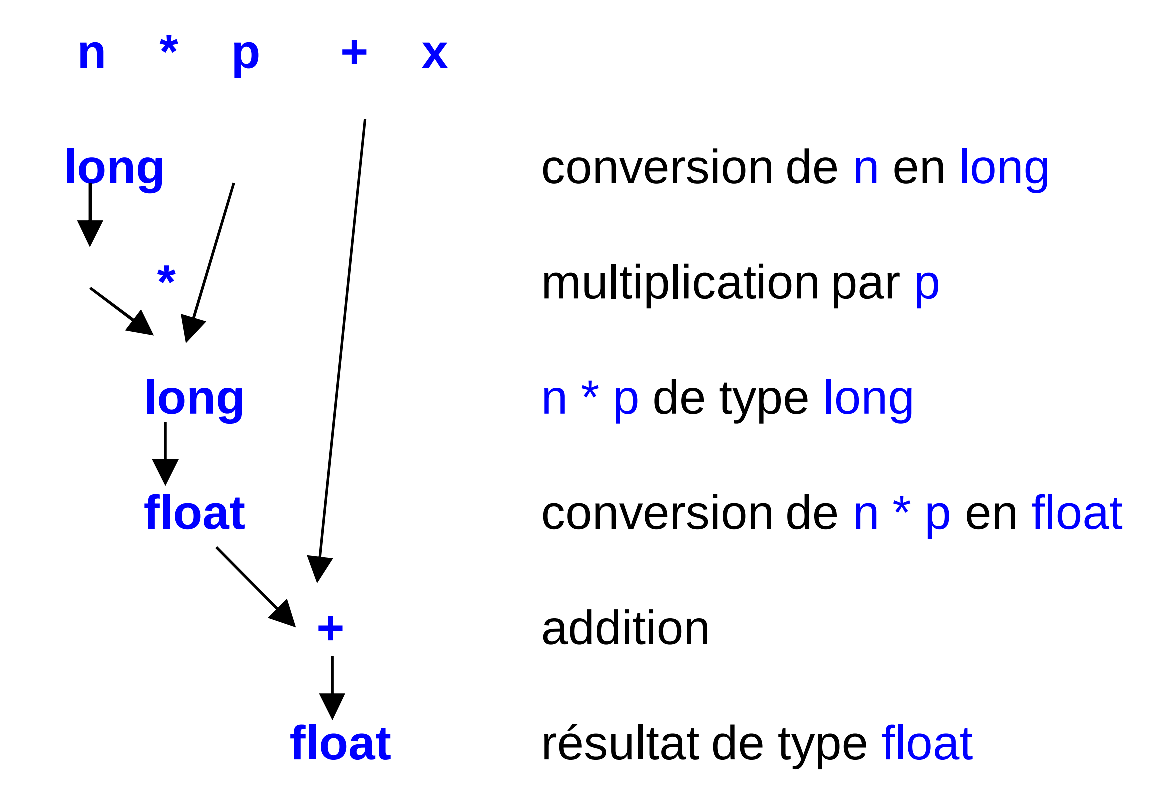
- La conversion se fait en général selon une hiérarchie qui n'altère pas les valeurs int → long → float → double → long double

- **Exemple1 :** n * x + p (int n,p; float x)
  - exécution prioritaire de n * x : conversion de n en float
  - exécution de l'addition : conversion de p en float

- **Exemple2 :** p1 * p2 + p3 * x (char p1, short p2, p3 ; float x)
  - p1, p2 et p3 d'abord convertis en int
  - p3 converti en float avant multiplication

Exemple de conversion

Exemple : n * p + x (int n ; long p ; float x)



conversion de n en long

multiplication par p

n * p de type long

conversion de n * p en float

addition

résultat de type float

Opérateur d’affectation simple =

- L’opérateur = affecte une valeur ou une expression à une variable
  - Exemple: double x,y,z; x=2.5; y=0.7; z=x*y-3;
- Le terme à gauche de l’affectation est appelé *lvalue* (left value)
- L'affectation est interprétée comme une expression. La valeur de l'expression est la valeur affectée
- On peut enchainer des affectations, l’évaluation se fait de droite à gauche
  - exemple : i = j = k= 5 (est équivalente à k = 5, j=k et ensuite i=j)
- La valeur affectée est toujours convertie dans le type de la lvalue, même si ce type est plus faible (ex : conversion de float en int, avec perte d'information)

Opérateurs relationnels

- **Opérateurs**
  - < : inférieur à
  - > : supérieur à
  - == : égal à
  - <= : inférieur ou égal à
  - >= : supérieur ou égal à
  - != : différent de

- Le résultat de la comparaison n’est pas une valeur booléenne, mais 0 si le résultat est faux et 1 si le résultat est vrai

- Les expressions relationnelles peuvent donc intervenir dans des expressions arithmétiques

- Exemple: a=2, b=7, c=4
  - b==3 ➔ 0 (faux)
  - a!=b ➔ 1(vrai)
  - 4*(a<b) + 2*(c>=b)➔ 4

Opérateurs logiques

- && : ET logique　　　　**|| :** OU logique　　　　! : négation logique
- && retourne vrai si les deux opérandes sont vrais (valent 1) et 0 sinon
- || retourne vrai si l'une des opérandes est vrai (vaut 1) et 0 sinon
- Les valeurs numériques sont acceptées : toute valeur non nulle correspond à vraie et 0 correspond à faux
  - Exemple : 5 && 11 ➔ 1
    !13.7 ➔ 0

Évaluation de && et ||

- Le 2ème opérande est évalué uniquement en cas de nécessité
  - a && b : b évalué uniquement si a vaut vrai (si a vaut faux, évaluation de b inutile car a && b vaut faux)
  - a || b : b évalué uniquement si a vaut faux (si a vaut vrai, évaluation de b inutile car a || b vaut vrai)

- **Exemples**
  - if ((d != 0) && (n / d == 2)) : pas de division si d vaut 0
  - if ((n >=0) && (sqrt(n) < p)) : racine non calculée si n < 0

- L'intérêt est d'accélérer l'évaluation et d'éviter les traitements inappropriés

Incrémentation et décrémentation

- Les opérateurs ++ et -- sont des opérateurs unaires permettant respectivement d'ajouter et de retrancher 1 au contenu de leur opérande
- Cette opération est effectuée après ou avant l'évaluation de l'expression suivant que l'opérateur suit ou précède son opérande
  - k = *i*++ *(post-incrémentation)* affecte d'abord la valeur de i à k et incrémente après (k = i++ ; ⇔ k = i ; i = i+1 ;)
  - k = ++*i (pré-incrémentation*) incrémente d'abord et après affecte la valeur incrémentée à k (k = ++i ; ⇔ i = i+1 ; k = i ;)
- Exemple : i = 5 ; n = ++i - 5 ;

  i vaut 6 et n vaut 1

  i = 5 ; n = i++ - 5 ;

  i vaut 6 et n vaut 0
- Remarque : idem pour l'opérateur de décrémentation --

Opérateurs de manipulations de bits

- **opérateurs arithmétiques bit à bit :**
  & : ET logique **| :** OU logique ^ : OU exclusif ~ : négation
- Les opérandes sont de type entier. Les opérations s'effectuent bit à bit suivant la logique binaire

b1	b2	~b1	b1&b2	b1 | b2	b1^b2
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

- Ex : 14= 1110 , 9=1001→ 14 & 9= 1000=8, 14 | 9 =1111=15

Opérateurs de décalage de bits

- Il existe deux opérateurs de décalage **:**

  >> : décalage à droite   **<<** : décalage à gauche

- L'opérande gauche constitue l'objet à décaler et l'opérande droit le nombre de bits de décalage

- Dans le cas d'un décalage à gauche les bits les plus à gauche sont perdus. Les positions binaires rendues vacantes sont remplies par des 0

  Ex : char x=14; (14=00001110) ➔ 14<<2 = 00111000 = 56

  char y=-7; (-7=11111001) ➔ -7<<2= 11100100**= -28**

- Rq : un décalage à gauche de k bits correspond (sauf débordement) à la multiplication par 2^k

Opérateurs de décalage de bits

- Lors d'un décalage à droite les bits les plus à droite sont perdus.
  - si l'entier à décaler est non signé, les positions binaires rendues vacantes sont remplies par des 0
  - s'il est signé le remplissage dépend de l'implémentation (en général le remplissage se fait par le bit du signe)

Ex : char x=14; (14=00001110) ➔ 14>>2 = **00**000011 = 3

- Remarque : un décalage à droite (n >> k) correspond à la division entière par 2^k si n est non signé

Opérateurs d’affectation combinés

- Soit un opérateur de calcul ***op*** et deux expressions *exp1* et *exp2*. L’expression *exp1= exp1 op exp2* peut s’écrire en général de façon équivalente sous la forme *exp1 op= exp2*

- Opérateurs utilisables :
  - += -= *= /= %=
  - <<= >>= &= ^= |=

- Exemples :
  - a=a+b s’écrit : a+=b
  - n=n%2 s’écrit : n%=2
  - x=x*i s’écrit : x*=i
  - p=p>>3 s’écrit : p>>=3

Opérateur de forçage de type (cast)

- Il est possible d’effectuer des conversions explicites ou de forcer le type d’une expression
  - Syntaxe : (<type>) <expression>
  - Exemple : int n, p ;
    (double) (n / p); convertit l’entier n / p en double

- Remarque : la conversion (ou casting) se fait après calcul

  (double) (n/p) ≠ (double) n / p ≠ (double) (n) / (double) (p)

  - float n = 4.6, p = 1.5 ;
    (int) n / (int) p = 4 / 1 = 4
    (int) n / p = 4 / 1.5 = 2.66
    n / (int) p = 4.6 / 1 = 4.6
    n / p = 4.6 / 1.5 = 3.06

Opérateur conditionnel ? :

- **Syntaxe:** **exp1 ? exp2 : exp3**

  exp1 est évaluée, si sa valeur est non nulle c'est exp2 qui est exécutée, sinon exp3

- **Exemple1 :** max = a > b ? a : b

  Si a>b alors on affecte à max le contenu de exp2 càd a sinon on lui affecte b

- **Exemple2 :** a>b ? i++ : i--;

  Si a>b on incrémente i sinon on décrémente i

Opérateur séquentiel ,

- Utilité : regrouper plusieurs sous-expressions ou calculs en une seule expression

- Les calculs sont évalués en séquence de gauche à droite

- La valeur de l'expression est celle de la dernière sous-expression

- Exemples
  - i++ , i + j; // on évalue i++ ensuite i+j (on utilise la valeur de i incrémentée)
  - i++ , j = i + k , a + b; // la valeur de l'expression est celle de a+b
  - for (i=1 , k=0 ; … ; …) { }

Opérateur SIZEOF

- **Syntaxe : sizeof (<type>) ou sizeof (<variable>)**

  fournit la taille en octets d'un type ou d'une variable

- **Exemples**
  - int n;
  - printf ("%d \n",sizeof(int)); // affiche 4
  - printf ("%d \n",sizeof(n)); // affiche 4

Priorité et associativité des opérateurs

- Une expression est évaluée en respectant des règles de priorité et d'associativité des opérateurs
  - Ex: * est plus prioritaire que +, ainsi 2 + 3 * 7 vaut 23 et non 35
- Le tableau de la page suivante donne la priorité de tous les opérateurs. La priorité est décroissante de haut en bas dans le tableau.
- Les opérateurs dans une même ligne ont le même niveau de priorité. Dans ce cas on applique les règles d'associativité selon le sens de la flèche. Par exemple: 13%3*4 vaut 4 et non 1
- Remarque: en cas de doute il vaut mieux utiliser les parenthèses pour indiquer les opérations à effectuer en priorité. Ex: (2 + 3) * 7 vaut 35

Priorités de tous les opérateurs

<table>
<tr><th>Catégorie</th><th>Opérateurs</th><th>Associativité</th></tr>
<tr><td>référence</td><td>() [] -> .</td><td>→</td></tr>
<tr><td>unaire</td><td>+ - ++ -- ! ~ * & (cast) sizeof</td><td>←</td></tr>
<tr><td>arithmétique</td><td>* / %</td><td>→</td></tr>
<tr><td>arithmétique</td><td>+ -</td><td>→</td></tr>
<tr><td>décalage</td><td><< >></td><td>→</td></tr>
<tr><td>relationnel</td><td>< <= > >=</td><td>→</td></tr>
<tr><td>relationnel</td><td>== !=</td><td>→</td></tr>
<tr><td>manip. de bits</td><td>&</td><td>→</td></tr>
<tr><td>manip. de bits</td><td>^</td><td>→</td></tr>
<tr><td>manip. de bits</td><td>|</td><td>→</td></tr>
<tr><td>logique</td><td>&&</td><td>→</td></tr>
<tr><td>logique</td><td>||</td><td>→</td></tr>
<tr><td>conditionnel</td><td>? :</td><td>→</td></tr>
<tr><td>affectation</td><td>= += -= *= /= %=</td><td>←</td></tr>
<tr><td></td><td>&= ^= |= <<= >>=</td><td>←</td></tr>
<tr><td>séquentiel</td><td>,</td><td>→</td></tr>
</table>

Cours Programmation I (chapitres 3&4)

Licence Fondamentale SMI (semestre 3)

Pr. Mouad BEN MAMOUN

ben_mamoun@fsr.ac.ma

Année universitaire 2014/2015

Chapitre 3

Entrées-sorties

Les instructions de lecture et d’écriture

- Il s’agit des instructions permettant à la machine de dialoguer avec l’utilisateur
  - Dans un sens la ***lecture*** permet à l’utilisateur d’entrer des valeurs au clavier pour qu’elles soient utilisées par le programme
  - Dans l’autre sens, ***l’écriture*** permet au programme de communiquer des valeurs à l’utilisateur en les affichant à l’écran (ou en les écrivant dans un fichier)
- La bibliothèque standard **<stdio>** contient un ensemble de fonctions qui assurent la lecture et l’écriture des données. Dans ce chapitre, nous allons en discuter les plus importantes:
  - **printf()** écriture formatée de données
  - **scanf()** lecture formatée de données

Ecriture formatée de données: printf ()

- la fonction **printf** est utilisée pour afficher à l'écran du texte, des valeurs de variables ou des résultats d'expressions.

- Syntaxe : **printf(**"format", expr1, expr2, …**);**
  - **expr1,…** : sont les variables et les expressions dont les valeurs sont à représenter
  - **Format** : est une chaîne de caractères qui peut contenir
    - du texte
    - des séquences d'échappement ('\n', '\t', …)
    - des spécificateurs de format : un ou deux caractères précédés du symbole ***%,*** indiquant le format d'affichage

Rq : Le nombre de spécificateurs de format doit être égale au nombre d'expressions!

Spécificateurs de format

SYMBOLE	TYPE	AFFICHAGE COMME
%d ou %i	*int*	entier relatif
%u	*unsinged int*	entier naturel non signé
%c	*char*	caractère
%o	*int*	entier sous forme octale
%x ou %X	*int*	entier sous forme hexadécimale
%f	*float, double*	réel en notation décimale
%e ou %E	*float, double*	réel en notation exponentielle
%s	*char**	chaîne de caractères

Séquences d’échappement

- l'affichage du texte peut être contrôlé à l'aide des *séquences d'échappement :*

  - ***\n*** : nouvelle ligne
  - ***\t*** : tabulation horizontale
  - ***\a*** : signal sonore
  - \b : retour arrière
  - \r : retour chariot
  - ***\v*** : tabulation verticale
  - ***\f*** : saut de page
  - **** : back slash (\ )
  - ***\’*** : apostrophe
  - ***\"*** : guillemet

Exemples de printf()

```
#include<stdio.h>
         main()
        {   int  i=1 , j=2, N=15;
           printf("la somme de %d et  %d est  %d \n", i, j, i+j);
           printf(« N= %x \n" , N);
           char c='A' ;
           printf(" le code Ascii de %c est  %d \n", c, c);
        }
```

Ce programme va afficher : *la somme de 1 et 2 est 3*

N=f

le code Ascii de A est 65

Remarque : Pour pouvoir traiter correctement les arguments du type long, il faut utiliser les spécificateurs %ld, %li, %lu, %lo, %lx

Exemples de printf()

```
#include<stdio.h>
        main()
        { double x=10.5, y=2.5;
          printf("%f  divisé par  %f  égal  à %f \n", x, y, x/y);
         printf("%e divisé par  %e égal  à %e\n", x, y, x/y);
         }
```

Ce programme va afficher :

10.500000 divisé par 2.500000 égal à 4.200000

1.050000e+001 divisé par 2.500000e+000 égal à   4.200000e+000

Remarque : Pour pouvoir traiter correctement les arguments du type long double, il faut utiliser les spécificateurs %lf et %le

Remarques sur l'affichage

- Par défaut, les entiers sont affichés sans espaces avant ou après
- Pour agir sur l'affichage ➔ un nombre est placé après % et précise le nombre de caractères **minimum à utiliser**
- Exemples : printf("**%4d**" , n);

n = 20	➔	~~20	(~ : espace)
n=56123	➔	56123	
printf("%4X", 123);	➔	~~7B	
printf("%4x", 123);	➔	~~7b	

Remarques sur l'affichage

- Pour les réels, on peut préciser la *largeur minimale* de la valeur à afficher et le nombre de chiffres après le point décimal.
- La précision par défaut est fixée à six décimales. Les positions décimales sont arrondies à la valeur la plus proche.
- Exemples :

```
printf("%f", 100.123);        ➔ 100.123000
printf("%12f", 100.123);      ➔ ~~100.123000
printf("%.2f", 100.123);      ➔ 100.12
printf("%5.0f", 100.123);     ➔  ~~100
printf("%10.3f", 100.123);    ➔  ~~~100.123
printf("%.4f", 1.23456);      ➔ 1.2346
```

Lecture formatée de données: scanf ()

- la fonction **scanf** permet de lire des données à partir du clavier

- Syntaxe : **scanf(**"format", AdrVar1, AdrVar2, …**);**
  - **Format** : le format de lecture de données, est le même que pour *printf*

  - ***adrVar1, adrVar2, …*** *:* adresses des variables auxquelles les données seront attribuées. L'adresse d'une variable est indiquée par le **nom** de la variable **précédé** du signe **&**

Exemples de scanf()

```
#include<stdio.h>
    main()
    {  int  i , j;
       scanf("%d%d", &i, &j);
       printf("i=%d  et  j=%d", i, j);
    }
```

ce programme permet de lire deux entiers entrés au clavier et les afficher à l’écran.

Remarque : pour lire une donnée du type **long**, il faut utiliser les spécificateurs **%ld**, **%li**, **%lu**, **%lo**, **%lx**.

Exemples de scanf()

```
#include<stdio.h>
main()
{ float x;
  double y;
  scanf("%f %lf", &x, &y);
  printf("x=%f  et  y=%f", x,y);
}
```

ce programme permet de lire un réel simple et un autre double du clavier et les afficher à l'écran

Remarque : pour lire une donnée du type **double**, il faut utiliser **%le** ou **%lf** et pour lire une donnée du type **long double**, il faut utiliser **%Le** ou **%Lf**

Chapitre 4

Structures de contrôle

Structures de contrôle

- Les structures de contrôle définissent la façon avec laquelle les instructions sont effectuées. Elles conditionnent l'exécution d'instructions à la valeur d'une expression

- On distingue :
  - **Les structures alternatives (tests) :** permettent d'effectuer des choix càd de se comporter différemment suivant les circonstances (valeur d'une expression). En C, on dispose des instructions : ***if…else*** et ***switch.***

  - **Les structures répétitives (boucles) :** permettent de répéter plusieurs fois un ensemble donné d'instructions. Cette famille dispose des instructions : **w*hile, do*…while** et ***for.***

L’instruction if...else

- Syntaxe : ***If*** *(expression)*
  bloc-instruction1
  else
  bloc-instruction2

  - *bloc-instruction* peut être une seule instruction terminée par un point-virgule ou une suite d’instructions délimitées par des accolades { }

  - *expression* est évaluée, si elle est vraie (valeur différente de 0), alors *bloc-instruction1* est exécuté. Si elle est fausse (valeur 0) alors *bloc-instruction2* est exécuté

- La partie *else* est facultative. S’il n’y a pas de traitement à réaliser quand la condition est fausse, on utilisera simplement la forme :
  If *(expression)* *bloc-instruction1*

if…else : exemples

- ```
  float a, b, max;
    if (a > b)
            max = a;
    else
            max = b;
  ```

- ```
  int a;
      if ((a%2)==0)
         printf(" %d est paire" ,a);
      else
         printf(" %d est impaire ",a);
  ```

Imbrication des instructions if

- On peut imbriquer plusieurs instructions if…else

- Ceci peut conduire à des confusions, par exemple :
  - ```
    if (N>0)
          if (A>B)
                MAX=A;
    else MAX=B;
    ```
    (interprétation 1 : si N=0 alors MAX prend la valeur B)
  - ```
    if (N>0)
          if (A>B)
                MAX=A;
          else MAX=B;
    ```
    (interprétation 2 : si N=0 MAX ne change pas)

- En C un *else* est toujours associé au dernier *if* qui ne possède pas une partie *else* (c'est l'interprétation 2 qui est juste)

Imbrication des instructions if

- Conseil : pour éviter toute ambiguïté ou pour forcer une certaine interprétation dans l'imbrication des ***if***, il vaut mieux utiliser les accolades

- ```
  if(a<=0)
        {if(a==0)
            printf("a est nul ");
         else
            printf(" a est strictement négatif ");}
  else
      printf(" a est strictement positif " );
  ```

- Pour forcer l'interprétation 1:
  ```
  if (N>0)
       { if (A>B)
            MAX=A;
       }
  else MAX=B;
  ```
  ```

L'instruction d'aiguillage switch :

- Permet de choisir des instructions à exécuter selon la valeur d'une expression qui doit être de type entier
- la syntaxe est :

switch *(expression) {*

*   **case** expression_constante1 : instructions_1;   **break;***

*   **case** expression_constante2 : instructions_2; **break**;*

*   …*

*   **case** expression_constante n : instructions_n; **break**;*

*   **default** : instructions;*

* }*

- *expression_constantei* doit être une expression constante **entière**
- Instructions i peut être une instruction simple ou composée
- *break* et *default sont* optionnels et peuvent ne pas figurer

Fonctionnement de switch

- *expression est évaluée*
- *si sa valeur est égale à une expression_constante i, on se branche à ce cas et on exécute les instructions_*i qui lui correspondent
  - On exécute aussi les instructions des cas suivants jusqu'à la fin du bloc ou jusqu'à une instruction break (qui fait sortir de la structure switch)
- si la valeur de l'expression n'est égale à aucune des expressions constantes
  - Si **default** existe, alors on exécute les instructions qui le suivent
  - Sinon aucune instruction n'est exécutée

Switch : exemple

```
main( )
{ char c;
    switch (c) {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
            case 'y': printf("voyelle\n"); break ;
            default : printf("consonne\n");
            }
}
```

Les boucles while et do .. while

```
while (condition)              do
   {                              {
       instructions                     instructions

   }                              } while (condition);
```

- la condition (dite condition de contrôle de la boucle) est évaluée à chaque itération. Les instructions (corps de la boucle) sont exécutées tant que la condition est vraie, on sort de la boucle dès que la condition devient fausse
- dans la boucle while le test de continuation s'effectue avant d'entamer le corps de boucle qui, de ce fait, peut ne jamais s'exécuter
- par contre, dans la boucle do-while ce test est effectué après le corps de boucle, lequel sera alors exécuté au moins une fois

Boucle while : exemple

Un programme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100

```
main( )
{       int  i, som;
         i =0; som= 0;
        while (som <=100)
                { i++;
                som+=i;
                }
        printf (" La valeur cherchée est N= %d\n ", i);
}
```

Boucle do .. while : exemple

Contrôle de saisie d'une note saisie au clavier jusqu'à ce que la valeur entrée soit valable (comprise entre 0 et 20)

```
main()
  { int N;
        do {
              printf (" Entrez une note comprise entre 0 et 20 \n");
              scanf("%d",&N);
            } while (N < 0 || N > 20);
  }
```

La boucle for

```
for (expr1 ; expr2 ; expr3)
    {
        instructions
    }
```

- L'expression expr1 est évaluée une seule fois au début de l'exécution de la boucle. Elle effectue l'initialisation des données de la boucle
- L'expression expr2 est évaluée et testée avant chaque passage dans la boucle. Elle constitue le test de continuation de la boucle.
- L'expression expr3 est évaluée après chaque passage. Elle est utilisée pour réinitialiser les données de la boucle

Boucle for : remarques

for (expr1 ; expr2 ; expr3) équivaut à :

```
for (expr1 ; expr2 ; expr3)
  {
    instructions
  }
```

```
 expr1;
while(expr2)
{ instructions
   expr3;
}
```

- En pratique, expr1 et expr3 contiennent souvent plusieurs initialisations ou réinitialisations, *séparées par des virgules*

Boucle for : exemple

Calcul de x à la puissance n où x est un réel non nul et n un entier positif ou nul

```
main ( )
{       float x, puiss;
         int n, i;
        {  printf (" Entrez respectivement les valeurs de x et n \n");
           scanf ("%f %d" , &x, &n);
           for (puiss =1, i=1; i<=n; i++)
                      puiss*=x;
           printf (" %f à la puissance %d  est égal à : %f", x,n,puiss);
        }
}
```

L’instruction break

- L’instruction break peut être utilisée dans une boucle (for, while, ou do .. while). Elle permet d’arrêter le déroulement de la boucle et le passage à la première instruction qui la suit

- En cas de boucles imbriquées, break ne met fin qu’ à la boucle la plus interne
-

```
{int i,j;
  for(i=0;i<4;i++)
    for (j=0;j<4;j++)
       { if(j==1) break;
         printf("i=%d,j=%d\n ",i,j);
       }
}
```

résultat:

i=0,j=0
i=1,j=0
i=2,j=0
i=3,j=0

L'instruction continue

- L'instruction continue peut être utilisée dans une boucle (for, while, ou do .. while). Elle permet l'abandon de l'itération courante et le passage à l'itération suivante

-
```
{int i;
   for(i=1;i<5;i++)
      {printf("début itération %d\n " ,i);
            if(i<3) continue;
            printf("  fin itération %d\n " ,i);
      }
  }
```

résultat:

début itération 1
début itération 2
début itération 3
fin itération 3
début itération 4
fin itération 4

Cours Programmation I (chapitres 5&6)

Licence Fondamentale SMI (semestre 3)

Pr. Mouad BEN MAMOUN

ben_mamoun@fsr.ac.ma

Année universitaire 2014/2015

Chapitre 5

Les tableaux

Exemple introductif

- Supposons qu'on veut conserver les notes d'une classe de 30 étudiants pour extraire quelques informations. Par exemple : calcul du nombre d'étudiants ayant une note supérieure à 10

- Le seul moyen dont nous disposons actuellement consiste à déclarer 30 variables, par exemple **N1, …, N30**. Après 30 instructions de saisie, on doit écrire 30 instructions if pour faire le calcul

```
nbre = 0
if (N1  >10)  nbre++ ;
….
if (N30>10)  nbre++ ;
```

c'est lourd à écrire

- Heureusement, les langages de programmation offrent la possibilité de rassembler toutes ces variables dans **une seule structure de donnée** appelée **tableau**

Tableaux

- Un **tableau** est une variable structurée composée d'un nombre de variables simples de même type désignées par un seul identificateur

- Ces variables simples sont appelées *éléments ou composantes* du tableau, elles sont stockées en mémoire à des emplacements contigus (l'un après l'autre)

- Le type des éléments du tableau peut être :
  - simple : char, int, float, double, …
  - pointeur ou structure

- On peut définir des tableaux :
  - à une dimension (tableau unidimensionnel ou vecteur)
  - à plusieurs dimensions (tableau multidimensionnel)

Déclaration des tableaux

- La déclaration d'un tableau à une dimension s'effectue en précisant le type de ses éléments et sa dimension (le nombre de ses éléments) :
  - Syntaxe en C : **Type identificateur[dimension];**
  - Exemple : **float notes[30];**
- La déclaration d’un tableau permet de lui réserver un espace mémoire dont la taille (en octets) est égal à : dimension * taille du type
- ainsi pour :
  - short A[100]; // on réserve 200 octets (100* 2octets)
  - char mot[10]; // on réserve 10 octets (10* 1octet)

Initialisation à la déclaration

- On peut initialiser les éléments d'un tableau lors de la déclaration, en indiquant la liste des valeurs respectives entre accolades. Ex:
  - int A[5]   = {1, 2, 3, 4, 5};
  - float B[4] = {-1.5, 3.3, 7e-2, -2.5E3};
- Si la liste ne contient pas assez de valeurs pour toutes les composantes, les composantes restantes sont initialisées par zéro
  - Ex: short T[10]   = {1, 2, 3, 4, 5};
- la liste ne doit pas contenir plus de valeurs que la dimension du tableau. Ex: short T[3] = {1, 2, 3, 4, 5};➔ Erreur
- Il est possible de ne pas indiquer la dimension explicitement lors de l'initialisation. Dans ce cas elle est égale au nombre de valeurs de la liste. Ex: short T[ ] = {1, 2, 3, 4, 5};➔ tableau de 5 éléments

Accès aux composantes d'un tableau

- L'accès à un élément du tableau se fait au moyen de l'indice. Par exemple, **T[i]** donne la valeur de l'élément i du tableau T

- En langage C l'indice du premier élément du tableau est 0. L'indice du dernier élément est égal à la dimension-1

Ex: int T[5] = {9, 8, 7, 6, 5};➔
T[0]=9, T[1]=8, T[2]=7, T[3]=6, T[4]=5

Remarques:

- on ne peut pas saisir, afficher ou traiter un tableau en entier, ainsi on ne peut pas écrire printf(" %d",T) ou scanf(" %d",&T)

- On traite les tableaux élément par élément de façon répétitive en utilisant des boucles

Tableaux : saisie et affichage

- Saisie des éléments d’un tableau T d’entiers de taille n :

```
 for(i=0;i<n;i++)
      { printf ("Entrez T[%d ] \n ",i );
        scanf(" %d" , &T[i]);
      }
```

- Affichage des éléments d’un tableau T de taille n :

```
for(i=0;i<n;i++)
      printf (" %d \t",T[i]);
```

Tableaux : exemple

- Calcul du nombre d'étudiants ayant une note supérieure à 10 :

```
main ( )
    {       float notes[30];
             int nbre,i;
            for(i=0;i<30;i++)
                { printf ("Entrez notes[%d] \n ",i);
                  scanf(" %f" , &notes[i]);
                }
              nbre=0;
              for (i=0; i<30; i++)
                      if (notes[i]>10) nbre++;
              printf (" le nombre de notes > à 10 est égal à : %d", nbre);

    }
```

Exercice

- Ecrire un programme qui permet de saisir un tableau de 10 réels et une valeur donnée et d'afficher si cette valeur se trouve dans le tableau ou non.

Tableaux à plusieurs dimensions

On peut définir un tableau à n dimensions de la façon suivante:

- **Type Nom_du_Tableau[D1][D2]…[Dn];** où Di est le nombre d'éléments dans la dimension i

- **Exemple :** pour stocker les notes de 20 étudiants en 5 modules dans deux examens, on peut déclarer un tableau :

float notes[20][5][2];

(notes[i][j][k] est la note de l'examen k dans le module j pour l'étudiant i)

Tableaux à deux dimensions (Matrices)

- Syntaxe : Type nom_du_Tableau[nombre_lignes][nombre_colonnes];
- Ex: **short A[2][3];** On peut représenter le tableau A de la manière suivante :

A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]

- Un tableau à deux dimensions A[n][m] est à interpréter comme un tableau unidimensionnel de dimension n dont chaque composante A[i] est un tableau unidimensionnel de dimension m.
- Un tableau à deux dimensions A[n][m] contient n***** m composantes. Ainsi lors de la déclaration, on lui réserve un espace mémoire dont la taille (en octets) est égal à : n*m* taille du type

Initialisation à la déclaration d'une Matrice

- L'initialisation lors de la déclaration se fait en indiquant la liste des valeurs respectives entre accolades ligne par ligne

- Exemple :
  - **float A[3][4] = {{-1.5, 2.1, 3.4, 0}, {8e-3, 7e-5,1, 2.7 }, {3.1, 0, 2.5E4, -1.3E2}};**

    A[0][0]=-1.5 , A[0][1]=2.1, A[0][2]=3.4, A[0][3]=0
    A[1][0]=8e-3 , A[1][1]=7e-5, A[1][2]=1, A[1][3]=2.7
    A[2][0]=3.1 , A[2][1]=0, A[2][2]=2.5E4, A[2][3]=-1.3E2

- On peut ne pas indiquer toutes les valeurs: Les composantes manquantes seront initialisées par zéro

- Comme pour les tableaux unidimensionnels, Il est défendu d'indiquer trop de valeurs pour une matrice

Matrices : saisie et affichage

- Saisie des éléments d'une matrice d'entiers A[n][m] :

```
for(i=0; i<n; i++)
      for(j=0; j<m; j++)
    { printf ("Entrez la valeur de A[%d][%d] \n ",i,j);
      scanf(" %d" , &A[i][j]);
    }
```

- Affichage des éléments d'une matrice d'entiers A[n][m] :

```
for(i=0; i<n; i++)
   { for(j=0; j<m; j++)
     printf (" %d \t",A[i][j]);
     printf("\n");
   }
```

Exercice

- Ecrire un programme qui permet de :
  - Saisir une matrice de réels de 10 lignes et 20 colonnes et une valeur réelle x
  - Afficher l'indice de la première colonne qui contient x s'il se trouve dans la matrice, sinon le message que x n'est pas dans la matrice

Exercice

Ecrire un programme qui transfère les éléments d'une matrice d'entiers A[n][m] (dimensions maximales: 10 lignes et 10 colonnes) dans un tableau T à une dimension n*m.

Exemple

$$\begin{pmatrix} 2 & 0 & 1 \\ 7 & 3 & 5 \\ 9 & 1 & 4 \\ 8 & 5 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 2 & 0 & 1 & 7 & 3 & 5 & 9 & 1 & 4 & 8 & 5 & 0 \end{pmatrix}$$

Représentation d’un tableau en mémoire

- La déclaration d’un tableau provoque la réservation automatique par le compilateur d’une zone contiguë de la mémoire.

- La mémoire est une succession de cases mémoires. Chaque case est une suite de 8 bits (1 octet), identifiée par un numéro appelé **adresse**.
  (on peut voir la mémoire comme une armoire constituée de tiroirs numérotés. Un numéro de tiroir correspond à une adresse)

- Les adresses sont souvent exprimées en hexadécimal pour une écriture plus compacte et proche de la représentation binaire de l’adresse. Le nombre de bits d’adressage dépend des machines.

- En C, **l’opérateur &** désigne **adresse de**. Ainsi, printf(" adresse de a= %x ", &a) affiche l’adresse de la variable a en hexadécimal

Représentation d'un tableau à une dimension en mémoire

- En C, le nom d'un tableau est le représentant de l'adresse du premier élément du tableau (pour un tableau T: **T=&T[0]**)

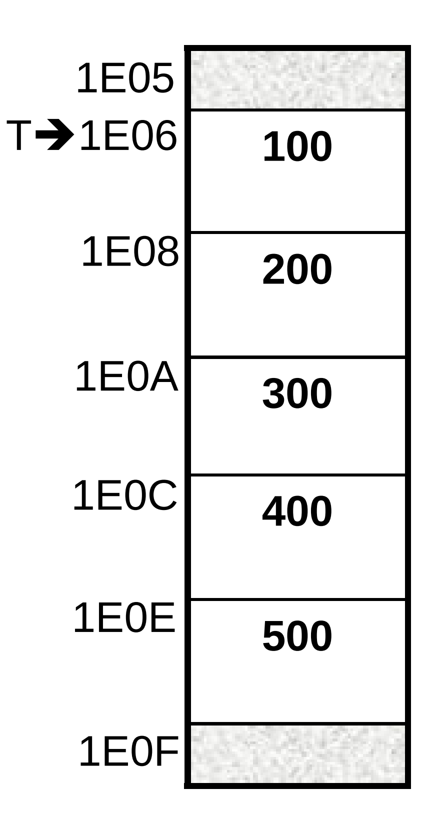
- Les composantes du tableau étant stockées en mémoire à des emplacements contigus, les adresses des autres composantes sont calculées (automatiquement) relativement à cette adresse :

  &T[i]= &T[0]+sizeof(type)*i

- Exemple : **short T[5] = {100, 200, 300, 400, 500};** et supposons que **T=&T[0]** =1E06

- On peut afficher et vérifier les adresses du tableau:

```
for(i=0;i<5;i++)
printf("adresse de T[%d]= %x\n",i,&T[i]);
```



Représentation d'un tableau à deux dimensions en mémoire

- Les éléments d'un tableau sont stockées en mémoire à des emplacements contigus ligne après ligne

- Comme pour les tableaux unidimensionnels, le nom d'un tableau A à deux dimensions est le représentant de l'adresse du premier élément : **A=&A[0][0]**

- Rappelons qu'une matrice A[n][m] est à interpréter comme un tableau de dimension n dont chaque composante A[i] est un tableau de dimension m.

A[i] et &A[i][0] représentent l'adresse du 1er élément de la ligne i (pour i de 0 à n-1)

- Exemple : **char A[3][4];** A=&A[0][0] =0118

A[0]→0118
A[1]→011C
A[2]→ 0120

A[0][0]
A[0][1]
A[0][2]
A[0][3]
A[1][0]
A[1][1]
A[1][2]
A[1][3]
A[2][0]
A[2][1]
A[2][2]
A[2][3]

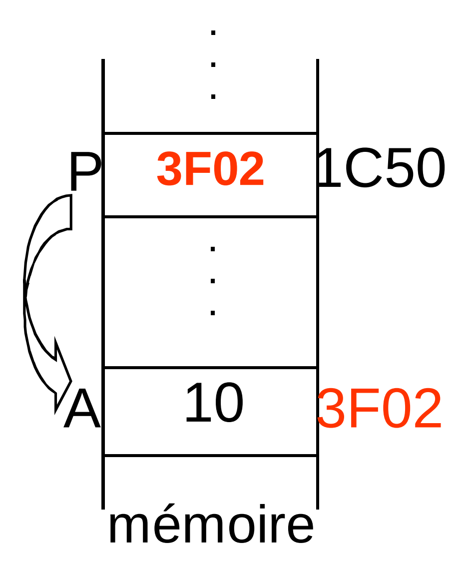
Chapitre 6

Les pointeurs

Pointeurs : définition

- Un pointeur est une variable spéciale qui peut contenir l'adresse d'une autre variable.

- Exemple : Soit A une variable contenant la valeur 10 et P un pointeur qui contient l'adresse de A *(on dit que P pointe sur A)* .



- Remarques :
  - Le nom d'une variable permet d'accéder *directement* à sa valeur (**adressage direct**).
  - Un pointeur qui contient l'adresse de la variable, permet d'accéder *indirectement* à sa valeur (**adressage indirect**).
  - Le nom d'une variable est lié à la même adresse, alors qu'un pointeur peut pointer sur différentes adresses

Intérêts des pointeurs

- Les pointeurs présentent de nombreux avantages :
  - Ils sont indispensables pour permettre le passage par référence pour les paramètres des fonctions
  - Ils permettent de créer des structures de données (listes et arbres) dont le nombre d'éléments peut évoluer dynamiquement. Ces structures sont très utilisées en programmation.
  - Ils permettent d'écrire des programmes plus compacts et efficaces

Déclaration d'un pointeur

- En C, chaque pointeur est limité à un type de donnée (même si la valeur d'un pointeur, qui est une adresse, est toujours un entier).
- Le type d'un pointeur dépend du type de la variable pointée. Ceci est important pour connaître la taille de la valeur pointée.
- On déclare un pointeur par l'instruction : **type *nom-du-pointeur ;**
  - type est le type de la variable pointée
  - ***** est l'opérateur qui indiquera au compilateur que c'est un pointeur
  - Exemple :

    int *pi; //pi est un pointeur vers une variable de type int

    float *pf; //pf est un pointeur vers une variable de type float
- Rq: la valeur d'un pointeur donne l'adresse du premier octet parmi les n octets où la variable est stockée

Opérateurs de manipulation des pointeurs

- Lors du travail avec des pointeurs, nous utilisons :
  - un **opérateur 'adresse de': &** pour obtenir l'adresse d'une variable
  - un **opérateur 'contenu de': *** pour accéder au contenu d'une adresse

- Exemple1 :
  - **int * p;** //on déclare un pointeur vers une variable de type int
  - **int i=10, j=30;** // deux variables de type int
  - **p=&i;** // on met dans p, l'adresse de i (p pointe sur i)
  - **printf("*p = %d \n",*p);** //affiche : *p = 10
  - ***p=20;** // met la valeur 20 dans la case mémoire pointée par p (i vaut 20 après cette instruction)
  - **p=&j;** // p pointe sur j
  - **i=*p;** // on affecte le contenu de p à i (i vaut 30 après cette instruction)

Opérateurs de manipulation des pointeurs

- Exemple2 :

```
float a, *p;
p=&a;
printf("Entrez une valeur : \n");
scanf("%f ",p); //supposons qu'on saisit la valeur 1.5
printf("Adresse de a= %x, contenu de a= %f\n" , p,*p);
*p+=0.5;
printf ("a= %f\n" , a); //affiche a=2.0
```

- **Remarque :** si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X
  - X+=2 équivaut à *P+=2
  - ++X équivaut à ++ *P
  - X++ équivaut à (*P)++ // les parenthèses ici sont obligatoires car l'associativité des opérateurs unaires * et ++ est de droite à gauche

Initialisation d'un pointeur

- A la déclaration d'un pointeur p, on ne sait pas sur quel zone mémoire il pointe. Ceci peut générer des problèmes :
  - **int *p;**
    ***p = 10;** //provoque un problème mémoire car le pointeur p n'a pas été initialisé

- **Conseil :** Toute utilisation d'un pointeur doit être précédée par une initialisation.

- On peut initialiser un pointeur en lui affectant :

  - l'adresse d'une variable (Ex: **int a, *p1; p1=&a;**)
  - un autre pointeur déjà initialisé (Ex: **int *p2; p2=p1;**)
  - la valeur 0 désignée par le symbole NULL, défini dans <stddef.h>.
    Ex: **int *p; p=0;ou p=NULL;** (on dit que p pointe 'nulle part': aucune adresse mémoire ne lui est associé)

- Rq: un pointeur peut aussi être initialisé par une allocation dynamique (voir fin du chapitre)

Pointeurs : exercice

```
main()
 {  int A = 1, B = 2, C = 3, *P1, *P2;
    P1=&A;
    P2=&C;
    *P1=(*P2)++;
     P1=P2;
     P2=&B;
    *P1-=*P2;
     ++*P2;
    *P1*=*P2;
     A=++*P2**P1;
     P1=&A;
    *P2=*P1/=*P2;
 }
```

Donnez les valeurs de A, B,C,P1 et P2 après chaque instruction

Opérations arithmétiques avec les pointeurs

- La valeur d'un pointeur étant un entier, certaines opérations arithmétiques sont possibles : ajouter ou soustraire un entier à un pointeur ou faire la différence de deux pointeurs

- Pour un entier i et des pointeurs p, p1 et p2 sur une variable de type T
  - **p+i (resp p-i)** : désigne un pointeur sur une variable de type T. Sa valeur est égale à celle de p incrémentée (resp décrémentée) de i*sizeof(T).
  - **p1-p2 :** Le résultat est un entier dont la valeur est égale à (différence des adresses)/sizeof(T).
- Remarque:
  - on peut également utiliser les opérateurs ++ et -- avec les pointeurs
  - la somme de deux pointeurs n'est pas autorisée

Opérations arithmétiques avec les pointeurs

Exemple :

```
float *p1, *p2;
float z =1.5;
p1=&z;
printf("Adresse p1 = %x \n",p1);
p1++;
p2=p1+1;
printf("Adresse p1 = %x \t Adresse p2 = %x\n",p1,p2);
printf("p2-p1 = %d \n",p2-p1);
```

Affichage :

Adresse p1 = 22ff44

Adresse p1 = 22ff48   Adresse p2 = 22ff4c

p2-p1=1

Pointeurs et tableaux

- Comme on l'a déjà vu au chapitre 5, le nom d'un tableau T représente l'adresse de son premier élément (T=&T[0]). Avec le formalisme pointeur, on peut dire que T est un **pointeur constant** sur le premier élément du tableau.

- En déclarant un tableau T et un pointeur P du même type, l'instruction P=T fait pointer P sur le premier élément de T (P=&T[0]) et crée une liaison entre P et le tableau T.

- A partir de là, on peut manipuler le tableau T en utilisant P, en effet :
  - **P pointe sur T[0] et *P désigne T[0]**
  - **P+1 pointe sur T[1] et *(P+1) désigne T[1]**
  - **….**
  - **P+i pointe sur T[i] et *(P+i) désigne T[i]**

Pointeurs et tableaux : exemple

- Exemple: **short x, A[7]={5,0,9,2,1,3,8};**

```
short *P;
P=A;
x=*(P+5);
```

- Le compilateur obtient l'adresse P+5 en ajoutant 5*sizeof(short) = 10 octets à l'adresse dans P
- D’autre part, les composantes du tableau sont stockées à des emplacements contigus et &A[5]= &A[0]+sizeof(short)*5=A+10
- Ainsi, x est égale à la valeur de A[5] (x= A[5])

Pointeurs : saisie et affichage d'un tableau

Version 1:

```
main()
 { float T[100] , *pt;
   int i,n;
   do {printf("Entrez n \n " );
         scanf(" %d" ,&n);
        }while(n<0 ||n>100);

   pt=T;
   for(i=0;i<n;i++)
        { printf ("Entrez T[%d] \n ",i );
           scanf(" %f" , pt+i);
         }

    for(i=0;i<n;i++)
           printf (" %f \t",*(pt+i));
}
```

Version 2: sans utiliser i

```
main()
 { float T[100] , *pt;
   int n;
   do {printf("Entrez n \n " );
         scanf(" %d" ,&n);
        }while(n<0 ||n>100);


    for(pt=T;pt<T+n;pt++)
{ printf ("Entrez T[%d] \n ",pt-T );
           scanf(" %f" , pt);
        }

    for(pt=T;pt<T+n;pt++)
           printf (" %f \t",*pt);
 }
```

Pointeurs et tableaux : exercice

- Soit P un pointeur qui 'pointe' sur un tableau A:

  int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};

  int *P; P = A;

- Quelles valeurs ou adresses fournissent ces expressions:
  - **a) *P+2**
  - **b) *(P+2)**
  - **c) &A[4]-3**
  - **d) A+3**
  - **e) &A[7]-P**
  - **f) P+(*P-10)**
  - **g) *(P+*(P+8)-A[7])**

Pointeurs et tableaux : exercice

- Soit P un pointeur qui 'pointe' sur un tableau A:

  int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};

  int *P; P = A;

- Quelles valeurs ou adresses fournissent ces expressions:
  - **a) *P+2** => la valeur 14
  - **b) *(P+2)** => la valeur 34
  - **c) &A[4]-3** => l'adresse de la composante A[1]
  - **d) A+3** => l'adresse de la composante A[3]
  - **e) &A[7]-P** => la valeur (indice) 7
  - **f) P+(*P-10)** => l'adresse de la composante A[2]
  - **g) *(P+*(P+8)-A[7])** =>la valeur 23

Pointeurs et tableaux à deux dimensions

- Le nom d'un tableau A à deux dimensions est un pointeur constant sur le premier élément du tableau càd A[0][0].

- En déclarant un tableau A[n][m] et un pointeur P du même type, on peut manipuler le tableau A en utilisant le pointeur P en faisant pointer P sur le premier élément de A (P=&A[0][0]), Ainsi :
  - **P pointe sur A[0][0] et *P désigne A[0][0]**
  - **P+1 pointe sur A[0][1] et *(P+1) désigne A[0][1]**
  - **….**
  - **P+M pointe sur A[1][0] et *(P+M) désigne A[1][0]**
  - **….**
  - **P+i*M pointe sur A[i][0] et *(P+i*M) désigne A[i][0]**
  - **….**
  - **P+i*M+j pointe sur A[i][j] et *(P+i*M+j) désigne A[i][j]**

Pointeurs : saisie et affichage d'une matrice

```
#define N 10
#define M 20
main( )
 { int i, j, A[N][M], *pt;
   pt=&A[0][0];
   for(i=0;i<N;i++)
       for(j=0;j<M;j++)
       { printf ("Entrez A[%d][%d]\n ",i,j );
          scanf(" %d" , pt+i*M+j);
        }

   for(i=0;i<N;i++)
      { for(j=0;j<M;j++)
           printf (" %d \t",*(pt+i*M+j));
        printf ("\n");
      }
 }
```

Pointeurs et tableaux : remarques

En C, on peut définir :

- **Un tableau de pointeurs :**

  Ex : **int *T[10];** //déclaration d'un tableau de 10 pointeurs d'entiers

- **Un pointeur de tableaux :**

  Ex : **int (*pt)[20];** //déclaration d'un pointeur sur des tableaux de 20 éléments

- **Un pointeur de pointeurs :**

  Ex : **int **pt;** //déclaration d'un pointeur pt qui pointe sur des pointeurs d'entiers

Allocation dynamique de mémoire

- Quand on déclare une variable dans un programme, on lui réserve implicitement un certain nombre d'octets en mémoire. Ce nombre est connu avant l'exécution du programme

- Or, il arrive souvent qu'on ne connaît pas la taille des données au moment de la programmation. On réserve alors l'espace maximal prévisible, ce qui conduit à un gaspillage de la mémoire

- Il serait souhaitable d'allouer la mémoire en fonction des données à saisir (par exemple la dimension d'un tableau)

- Il faut donc un moyen pour allouer la mémoire lors de l'exécution du programme : c'est l'allocation dynamique de mémoire

La fonction malloc

- La fonction **malloc** de la bibliothèque <*stdlib*> permet de localiser et de réserver de la mémoire, sa syntaxe est : **malloc(N)**

- Cette fonction retourne un pointeur de type char * pointant vers le premier octet d'une zone mémoire libre de N octets ou le pointeur NULL s'il n'y a pas assez de mémoire libre à allouer.

- Exemple : Si on veut réserver la mémoire pour un texte de 1000 caractères, on peut déclarer un pointeur p sur **char** (**char *p**).
  - L'instruction: **p = malloc(1000);** fournit l'adresse d'un bloc de 1000 octets libres et l'affecte à p. S'il n'y a pas assez de mémoire, p obtient la valeur zéro (NULL).

- Remarque : Il existe d'autres fonctions d'allocation dynamique de mémoire dans la bibliothèque <*stdlib*>

La fonction malloc et free

- Si on veut réserver de la mémoire pour des données qui ne sont pas de type char, il faut convertir le type de la sortie de la fonction malloc à l'aide d'un cast.

- Exemple : on peut réserver la mémoire pour 2 variables contiguës de type int avec l'instruction : p = (int*)malloc(2 * sizeof(int)); où p est un pointeur sur **int** (**int *p**).

- Si on n'a plus besoin d'un bloc de mémoire réservé par **malloc**, alors on peut le libérer à l'aide de la fonction **free** , dont la synatxe est : **free(pointeur)**;

- Si on ne libère pas explicitement la mémoire à l'aide de **free**, alors elle est libérée automatiquement à la fin du programme.

malloc et free : exemple

Saisie et affichage d'un tableau

```
#include<stdio.h>
#include<stdlib.h>
main()
 { float  *pt;
   int i,n;
   printf("Entrez la taille du tableau \n" );
   scanf(" %d" ,&n);

   pt=(float*) malloc(n*sizeof(float));
   if (pt==Null)
    {
     printf( " pas assez de mémoire \n" );
     system(" pause " );
    }
printf(" Saisie du tableau \n " );
for(i=0;i<n;i++)
  { printf ("Élément  %d ? \n ",i+1);
    scanf(" %f" , pt+i);
   }

printf(" Affichage du tableau \n " );
for(i=0;i<n;i++)
     printf (" %f \t",*(pt+i));
free(pt);
}
```

Exercice

- Ecrire un programme qui permet d'allouer la mémoire dynamiquement à un tableau T de n réels, de saisir le tableau T et une valeur donnée x. Ensuite, le programme affiche si cette valeur se trouve dans le tableau ou non. Utilisez uniquement le formalisme pointeur.

Cours Programmation I (chapitre 7)

Licence Fondamentale SMI (semestre 3)

Pr. Mouad BEN MAMOUN

ben_mamoun@fsr.ac.ma

Année universitaire 2014/2015

Chapitre 7

Les fonctions

La programmation modulaire

- Certains problèmes conduisent à des programmes longs, difficiles à écrire et à comprendre. On les découpe en des parties appelées **sous-programmes** ou **modules**
- Les modules sont des groupes d'instructions qui fournissent une solution à des parties bien définies d'un problème plus complexe. Ils ont plusieurs **intérêts** :
  - permettent de **"factoriser" les programmes**, càd de mettre en commun les parties qui se répètent
  - permettent une **structuration** et une **meilleure lisibilité** des programmes
  - **facilitent la maintenance** du code (il suffit de modifier une seule fois)
  - peuvent éventuellement être **réutilisées** dans d'autres programmes
- La structuration de programmes en sous-programmes se fait en C à l'aide des **fonctions**

Fonctions

- On définit une fonction par la syntaxe suivante:

  type nom_fonction (**type1** arg1,…, **typeN** argN)

  {

  instructions constituant le corps de la fonction

  return (expression)

  }

- Dans la première ligne (appelée **en-tête de la fonction**) :
  - **type** est le type du résultat retourné. Si la fonction n'a pas de résultat à retourner, elle est de type **void.**
  - le choix d'un nom de fonction doit respecter les mêmes règles que celles adoptées pour les noms de variables.
  - entre parenthèses, on spécifie les **arguments** de la fonction et leurs types. Si une fonction n'a pas de paramètres, on peut déclarer la liste des paramètres comme **(void)** ou simplement comme **()**

- Pour fournir un résultat en quittant une fonction, on dispose de la commande **return.**

Fonctions : exemples

- Une fonction qui calcule la somme de deux réels x et y :

```
double Som(double x, double y )
    {
       return (x+y);
    }
```

- Une fonction qui affiche la somme de deux réels x et y :

```
void AfficheSom(double x, double y)
   {
      printf (" %lf", x+y );
   }
```

- Une fonction qui renvoie un entier saisi au clavier

```
int RenvoieEntier( void )
    {
       int n;
       printf (" Entrez n \n");
       scanf (" %d ", &n);
       return n;
    }
```

- Une fonction qui affiche les éléments d'un tableau d'entiers

```
void AfficheTab(int T[ ], int n)
    {  int i;
        for(i=0;i<n;i++)
        printf (" %d \t", T[i]);
    }
```

Appel d’une fonction

- L‘appel d'une fonction se fait par simple écriture de son nom avec la liste des paramètres : nom_fonction (para1,…, paraN)
- Lors de l'appel d'une fonction, les paramètres sont appelés **paramètres effectifs** : ils contiennent les valeurs pour effectuer le traitement. Lors de la définition, les paramètres sont appelés **paramètres formels**.
- L'ordre et les types des paramètres effectifs doivent correspondre à ceux des paramètres formels
- **Exemple d’appels:**

```
main( )
    {   double z;
        int A[5]   = {1, 2, 3, 4, 5};
       z=Som(2.5, 7.3);
       AfficheTab(A,5);
    }
```

Déclaration des fonctions

- Il est nécessaire pour le compilateur de connaître la définition d'une fonction au moment où elle est appelée.

- Si une fonction est définie après son premier appel (en particulier si elle définie après main), elle doit être **déclarée** auparavant.

- La déclaration d'une fonction se fait par son **prototype** qui indique les types de ses paramètres et celui de la fonction :

  type nom_fonction (**type1**,…, **typeN**)

Déclaration des fonctions : exemple

```
#include<stdio.h>
float ValeurAbsolue(float); //prototype de la fonction ValeurAbsolue
main( )
 { float  x=-5.7,y;
   y= ValeurAbsolue(x);
    printf("La valeur absolue de %f est : %f \n " , x,y);
 }
//Définition de la fonction  ValeurAbsolue
 float ValeurAbsolue(float a)
   {
       if (a<0) a=-a;
       return a;
   }
```

Variables locales et globales

- On peut manipuler 2 types de variables dans un programme C : des **variables locales** et des **variables globales**. Elles se distinguent par ce qu'on appelle leur **portée** (leur "espace de visibilité", leur "durée de vie")

- Une variable définie à l'intérieur d'une fonction est une **variable locale**, elle n'est connue qu'à l'intérieur de cette fonction. Elle est créée à l'appel de la fonction et détruite à la fin de son exécution

- Une variable définie à l'extérieur des fonctions est une **variable globale**. Elle est définie durant toute l'application et peut être utilisée et modifiée par les différentes fonctions du programme.

Variables locales et globales : remarques

- Les variables déclarées au début de la fonction principale main ne sont pas des variables globales, mais elles sont locales à main

- Une variable locale cache la variable globale qui a le même nom

- Il faut utiliser autant que possible des variables locales. Ceci permet d'économiser la mémoire et d'assurer l'indépendance de la fonction

- En C, une variable déclarée dans un bloc d'instructions est uniquement visible à l'intérieur de ce bloc. C’est une variable locale à ce bloc, elle cache toutes les variables du même nom des blocs qui l'entourent

Variables locales et globales : exemple

```
#include<stdio.h>
int x = 7;
int f(int);
int g(int);
main( )
{   printf("x = %d\t", x);
    {  int x = 6;   printf("x = %d\t", x); }
    printf("f(%d) = %d\t", x, f(x));
    printf("g(%d) = %d\t", x, g(x));
 }
int f(int a)  { int x = 9;   return (a + x); }
int g(int a) { return (a * x); }
```

Qu'affiche ce programme?

x=7   x=6   f(7)=16   g(7) = 49

Variables locales et globales : exemple

```
#include<stdio.h>
void f(void);
int i;
main( )
{ int k = 5;
  i=3; f(); f();
 printf("i = %d et k=%d \n", i,k);  }
void f(void)  { int k = 1;
                printf("i = %d et k=%d \n", i,k);
                i++;k++;}
```

Qu’affiche ce programme?

i=3 et k=1

i=4 et k=1

i=5 et k=5

Paramètres d'une fonction

- Les paramètres servent à échanger des informations entre la fonction appelante et la fonction appelée. Ils peuvent recevoir des données et stocker des résultats

- Il existe deux modes de transmission de paramètres dans les langages de programmation :

  - **La transmission par valeur** : les valeurs des paramètres effectifs sont affectées aux paramètres formels correspondants au moment de l'appel de la fonction ou procédure. *Dans ce mode, le paramètre effectif ne subit aucune modification*

  - **La transmission par adresse (ou par référence)** : les adresses des paramètres effectifs sont transmises à la fonction appelante. *Dans ce mode, le paramètre effectif subit les mêmes modifications que le paramètre formel*

Transmission des paramètres en C

- La transmission des paramètres en C se fait toujours par valeur
- Pour effectuer une transmission par adresse en C, on déclare le paramètre formel de type pointeur et lors d'un appel de la fonction, on envoie l'adresse et non la valeur du paramètre effectif
- Exemple :

```
void Increment (int x,  int *y)
    { x=x+1;
     *y =*y+1; }
  main( )
    { int n = 3, m=3;
      Increment (n, &m);
      printf("n = %d et m=%d \n", n,m);  }
```

Résultat :

n=3 et m= 4

Exemples

Une fonction qui échange le contenu de deux variables :

```
void Echange (float *x,  float *y)
    { float z;
          z = *x;
         *x = *y;
         *y = z;
   }
main()
    { float a=2,b=5;
      Echange(&a,&b);
      printf("a=%f,b=%f\n ",a,b);
   }
```

Exemples

Une fonction qui calcule la somme et le produit de deux réels:

```
void SommeProduit (float  x,  float  y, float *s, float *p)
   {      *s = x+y;
          *p = x*y;
   }

main()
   { float a=2.5,b=5.3,som,prod;
     SommeProduit(a,b,&som,&prod);
     printf("la somme=%f,le produit=%f\n ",som,prod);
  }
```

Récursivité

- Une fonction qui fait appel à elle-même est une fonction **récursive**
- Toute fonction récursive doit posséder un cas limite (cas trivial) qui arrête la récursivité
- Exemple : Calcul du factorielle

```
int  fact (int n )
{     if (n==0)    /*cas trivial*/
                return (1);
      else
                return (n* fact(n-1) );
}
```

Remarque : l'ordre de calcul est l'ordre inverse de l'appel de la fonction

Fonctions récursives : exercice

- Ecrivez une fonction récursive (puis itérative) qui calcule le terme n de la suite de Fibonacci définie par : U(0)=U(1)=1
  U(n)=U(n-1)+U(n-2)

```
int  Fib (int n)
   {
         if (n==0 || n==1)
                  return (1);
         else
                  return ( Fib(n-1)+Fib(n-2));

   }
```

Fonctions récursives : exercice (suite)

- Une fonction itérative pour le calcul de la suite de Fibonacci :

```
int  Fib (int n)
   { int i, AvantDernier, Dernier, Nouveau;
        if (n==0 || n==1)  return (1);
        AvantDernier=1; Dernier =1;
        for (i=2; i<=n; i++)
          { Nouveau= Dernier+ AvantDernier;
            AvantDernier = Dernier;
            Dernier = Nouveau;
          }
        return (Nouveau);
   }
```

Remarque: la solution récursive est plus facile à écrire

Fonctions récursives : exercice

Ecrivez une fonction récursive qui affiche la valeur binaire d'un entier n.

```
void binaire(int n)
     {   if (n!=0)
          {   binaire(n/2);
              printf("%d",n%2);
          }
     }
main()
     {  int p;
      printf ("entrez un entier p\n");
     scanf("%d",&p);
      if (p==0)           printf("%d",0);
     else binaire(p);
     }
```

Complément du cours Langage C

Licence Fondamentale SMI (semestre 3)

Pr. Mouad BEN MAMOUN

ben_mamoun@fsr.ac.ma

Année universitaire 2013/2014

Chapitre 9

Types structures, unions et synonymes

Structures

- Une **structure** est un nouveau type de données constitué par un ensemble de variables (champs) qui peuvent être hétérogènes et de types différents

- La différence avec le type tableau est que les champs d'un tableau sont tous homogènes et du même type

- Les structures permettent de représenter des objets réels caractérisées par plusieurs informations, par exemple :

  - Une personne caractérisée par son nom (chaîne), son âge (entier), sa taille (réel), …

  - Une voiture caractérisée par sa marque (chaîne), sa couleur (chaîne), son année modèle(entier), …

Déclaration d’une structure

La déclaration d'une structure s'effectue en précisant le nom de la structure, ainsi que le nom et le type de ses champs :

- Syntaxe en C :

```
struct  nom_structure
     {  type 1 nom_champ1;
        type 2  nom_champ2;
        …
        type N  nom_champN;
     };
```

- Exemple :

```
struct Personne
     { char Nom[20];
        int Age;
        float taille;
     };
```

Rq: Le nom d’une structure n’est pas un nom de variable, c’est le nom du type ou modèle de la structure

Déclaration d'une variable structure

- La déclaration d'une structure ne réserve pas d'espace mémoire

- La réservation se fait quand on définit des variables correspondant à ce modèle de structure. Ceci peut se faire soit :
  - après la déclaration de la structure, par exemple :

```
struct  Personne p1, *p2, tab[10];
//p2 est une variable de type pointeur sur une structure Personne
// tab est une variable de type tableau de 10 éléments (de type Personne)
```

  - ou au moment de la déclaration de la structure

```
struct Personne
    { char Nom[20];
      int Age;
      float taille;
    } p1, *p2, tab[10];
```

Initialisation d'une structure

- Lors de la déclaration d'une variable structure, on peut initialiser ses champs avec une notation semblable à celle utilisée pour les tableaux en indiquant la liste des valeurs respectives entre accolades.

- Exemple :

```
struct date
{ unsigned short jour;
  char mois[10];
  unsigned short annee;
} ;

struct  date d1= {15,"Novembre", 2013};
```

Accès aux champs d'une structure

- L'accès à un champ d'une variable structure se fait par le nom de la variable suivi d'un point et du nom du champ

  nom_variable.nom_champ

  Exemple: p1.age représente le champ age de la variable structure p1

- Dans le cas d'une variable structure de type pointeur

  (ex : **struct Personne *p2**), on utilise en général l'opérateur **->** pour accéder aux champs

  nom_variable ->nom_champ (ex: p2->age)

- Remarque : Il est possible d'utiliser un point même dans le cas d'un pointeur, par exemple **(*p2).age** (les parenthèses sont ici indispensables)

Accès aux champs d'une structure : Exemple

Saisie et affichage d'une structure

- Soit la structure article définie par :

```
struct article
    { int numero; //un numéro qui identifie l'article
      char nom[20];
      int qte_stock; // quantité disponible en stock
      float prix;
    } ;
```

- Soit la déclaration de variables :

```
struct article art, *pt_art;
```

Saisissez les champs de la structure art, puis affichez son contenu

Exemple de Saisie et d'affichage d'une structure

```
/* Saisie des champs de la structure art */
printf ("Entrez respectivement les champs de l'article \n ");
scanf(" %d %s %d %f" , &art.numero, art.nom, &art.qte_stock, &art.prix);

/* Saisie des champs de la structure art */
printf (" Cet article a pour : \n ");
printf (" \t numéro : %d \n ", art.numero);
printf (" \t nom : %s \n ", art.nom);
printf (" \t quantité en stock : %d \n ", art.qte_stock);
printf (" \t prix : %f \n ", art.prix);
```

Refaites la saisie et l'affichage en utilisant le pointeur pt_art

Exemple de Saisie et d’affichage via un pointeur

Remarque : il faut d’abord initialiser le pointeur pt_art

```
/* Saisie des champs de la structure art */
printf ("Entrez respectivement les champs de l’article \n ");
scanf(" %d %s %d %f" , &pt_art->numero, pt_art->nom,
                              & pt_art->qte_stock, & pt_art->prix);

/* Affichage des champs de la structure art */
printf (" Cet article a pour : \n ");
printf (" \t numéro : %d \n ", pt_art->numero);
printf (" \t nom : %s \n ", pt_art->nom);
printf (" \t quantité en stock : %d \n ", pt_art.->qte_stock);
printf (" \t prix : %f \n ", pt_art->prix);
```

Composition des structures

- Les structures peuvent être composées de champs de n'importe quel type connu: types de base, pointeur, tableau ou structure.

- Exemple de structure comportant un tableau et une structure :

```
struct Etudiant
    { int code;
      char Nom[20];
      struct date date_naissance;
      float notes[8]; // notes de l'étudiant dans 8 modules
    } E1,E2;
```

- on peut écrire E1.date_naissance.annee pour accéder au champ annee de E1.date_naissance qui est de type date

- E2.notes[3] représente la note du module 4 de l'étudiant E2

Manipulation des structures

- Les structures sont manipulées en général champ par champ. Toutefois, la norme ANSI permet d'affecter une structure à une autre structure de même type, par exemple : struct Etudiant E1,E2;

  E2=E1; est une instruction valide qui recopie tous les champs de E1 dans les champs de E2

- Rq: Il n'est pas possible de comparer deux structures. Les instructions if(E1==E2) ou if(E1!=E2) ne sont pas permises

- L'opérateur & permet de récupérer l'adresse d'une variable structure (ex : struct Personne p1, *p2; **p2=&p1**)

- L'opérateur sizeof permet de récupérer la taille d'un type structure ou d'une variable structure (ex : sizeof (struct Personne) ou sizeof (p1))

Structures et fonctions

Une structure peut être utilisée comme argument d'une fonction et transmise par valeur (ou par adresse via un pointeur)

Exemple de transmission par valeur

```
struct  couple { int a;
                 float b;};
void zero (struct couple s)
  { s.a=0; s.b=0;
    printf(" %d %f \n ", s.a, s.b);
  }
main()
 { struct couple x;
   x.a=1;x.b=2.3;
   printf("avant: %d %f \n ", x.a, x.b);
   zero(x);
   printf("après: %d %f \n ", x.a, x.b);
 }
```

Exemple de transmission par adresse

```
struct  couple { int a;
                 float b;};
void zero (struct couple *s)
  { s->a=0; s->b=0;
    printf(" %d %f \n ", s->a, s->b);
  }
main()
 { struct couple x;
   x.a=1;x.b=2.3;
   printf("avant: %d %f \n ", x.a, x.b);
   zero(&x);
   printf("après: %d %f \n ", x.a, x.b);
 }
```

Structures et fonctions (2)

Une structure peut être retournée par l'instruction return d'une fonction

Exemple

```
struct  couple { int a;
                 float b;};
struct couple oppose(struct couple s)
   {struct  couple z;
     z.a=-s.a; z.b=-s.b;
    return z;
   }
```

```
main()
 { struct couple x,y;
   x.a=1;x.b=2.3;
   printf("x: %d %f \n ",x.a,x.b);
   y=oppose(x);
   printf("y: %d %f \n ",y.a,y.b);
 }
```

Structures et fonctions : exercice

Pour la structure article définie auparavant :

- Ecrivez une fonction, nommée SaisieArticle, qui saisit les champs d'une variable article passé en paramètre
- Ecrivez une fonction, nommée AfficheArticle, qui affiche le contenu des champs d'une variable article passé en paramètre
- Ecrivez une fonction nommée SaisieTabArticle, qui remplit un tableau T de n articles. T et n sont des paramètres de la fonction.
- Ecrivez une fonction AfficheTabStock, qui affiche les articles de T ayant une quantité en stock >= à une valeur q. T, n et q sont des paramètres de la fonction
- Ecrivez un programme qui fait appel aux fonctions SaisieTabArticle et AfficheTabStock

Structures et fonctions : exercice (2)

```
struct article
 {int numero;
  char nom[20];
  int qte_stock;
  float prix;
} ;

void SaisieArticle (struct article * art)
{printf (" numero ? \n");
scanf (" %d",&art->numero);
printf (" nom ? \n ");
scanf("%s",art->nom);
printf (" quantité en stock ? \n ") ;
scanf (" %d" , &art->qte_stock);
printf (" prix ?\n ");
scanf (" %f" ,&art->prix);
}
```

```
void  AfficheArticle (struct article  art)
{printf (" Cet article a pour : \n ");
printf (" \t numéro : %d \n ", art.numero);
printf (" \t nom : %s \n ", art.nom);
printf (" \t stock : %d \n ", art. qte_stock);
printf (" \t prix : %f \n ", art.prix);
}

void SaisieTabArticle(struct articleT[ ], int n)
 {int i;
  for(i=0;i<n;i++)
   {printf (" saisie de l'article %d \n " ,i+1);
    SaisieArticle (&T[i]);
   }
  }
```

Structures et fonctions : exercice (3)

```
void AfficheTabStock(struct article *T, int n, int q)
{  int i;
   printf (" Les articles ayant un stock   >=%d sont : \n " ,q);

  for(i=0;i<n;i++)
      if(T[i]. qte_stock>=q)
             AfficheArticle(T[i]);
}

main( )
 { struct articleT[10];
   SaisieTabArticle(T, 10);
   AfficheTabStock(T, 10, 2);
 }
```

Définition de types synonymes: typedef

- En C, on peut définir des types nouveaux synonymes de types existants (simples, pointeur, tableau, structure,…) en utilisant le mot clé **typedef**. Ces nouveaux types peuvent être utilisées ensuite comme les types prédéfinis

- **Exemple d'un type synonyme d'un type simple :**

typedef int entier; définit un nouveau type appelé **entier** synonyme du type prédéfini **int**

entier i=4,T[10] ; le type entier est utilisé ici pour déclarer des variables

Remarque : l'intérêt de typedef pour les types de base est limité puisqu'il remplace simplement un nom par un autre

- **Exemple d'un type synonyme d'un type pointeur :**

typedef int * ptr_entier;
ptr_entier p1,p2; p1 et p2 sont des pointeurs sur des int

typedef char* chaine;
chaine ch;

Exemples de typedef

- **Type synonyme d'un type tableau :**

typedef float tableau[10];
tableau T; T est un tableau de 10 réels
typedef int matrice[4][5];
matrice A; A est une matrice d'entiers de 4 lignes et 5 colonnes

- **Type synonyme d'un type structure :**

```
typedef struct
  {  int jour;
     int mois;
     int annee;
  } date;
```

- date est le nom du nouveau type et non d'une variable
- On peut utiliser directement ce type, par exemple: **date d, *p;**
- Ceci permet simplement d'éviter l'emploi du mot clé struct dans les déclarations de variables

Structures récursives

- Une structure est dite récursive si elle contient des pointeurs vers elle même, par exemple :

```
struct Individu
{ char *Nom;
  int Age;
 struct Individu *Pere, *Mere;
};
```

```
struct noeud
{  int val;
 struct noeud *suivant;
};
```

- Ce genre de structures est fondamental en programmation car il permet d'implémenter la plupart des structures de données employées en informatique (seront étudiées en cours de Structures de données)

Unions

- Une **union** est un objet défini par un ensemble de données (comme une structure) mais qui ne peut prendre qu'une seule donnée à la fois à un instant donné

- Une union se déclare comme une structure :
  - **union nom_union**

```
union  nom_union
  {  type 1 nom_champ1;
     type 2  nom_champ2;
     …
     type N  nom_champN;
  };
```

Exemple :

```
union IntFloat
{  int n;
   float x;
} A;
```

  - A un instant donné, la variable A contiendra un entier ou un réel, mais pas les deux !
  - C'est au programmeur de connaître à tout instant le type de l'information contenue dans A

Unions (2)

- Les unions permettent d'utiliser un même espace mémoire pour des données de types différents à des moments différents

- Lorsque l'on définit une variable correspondant à un type union, le compilateur réserve l'espace mémoire nécessaire pour stocker le champ de plus grande taille appartenant à l'union

- La syntaxe d'accès aux champs d'une union est identique à celle pour accéder aux champs d'une structure (dans l'exemple, Le programmeur utilisera A.n s'il désire manipuler l'information de type int, sinon il utilisera A.x)

- De façon générale, une union se manipule comme une structure